10-02-00

A

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventorship.....Alcazar et al.
Applicant.....Microsoft Corporation
Attorney's Docket No. MS1-657US
Title: Event Routing Model For An Extensible Editor

TRANSMITTAL LETTER AND CERTIFICATE OF MAILING

To: Commissioner of Patents and Trademarks,
Washington, D.C. 20231

From: James R. Banowsky (Tel. 509-324-9256; Fax 509-323-8979)
Lee & Hayes, PLLC
421 W. Riverside Avenue, Suite 500
Spokane, WA 99201

The following enumerated items accompany this transmittal letter and are being submitted for the matter identified in the above caption.

1. Specification–title page, plus 60 pages, including 47 claims and Abstract
2. Transmittal letter including Certificate of Express Mailing
3. 7 Sheets Formal Drawings (Figs. 1-7)
4. Return Post Card

Large Entity Status [x] Small Entity Status [ ]

Date: 9/29/00 By: James R. Banowsky
Reg. No. 37,773

CERTIFICATE OF MAILING

I hereby certify that the items listed above as enclosed are being deposited with the U.S. Postal Service as either first class mail, or Express Mail if the blank for Express Mail No. is completed below, in an envelope addressed to The Commissioner of Patents and Trademarks, Washington, D.C. 20231, on the below-indicated date. Any Express Mail No. has also been marked on the listed items.

Express Mail No. (if applicable) EL685271436

Date: 9/29/2000 By: Dana L Calhoun
Dana L. Calhoun

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Event Routing Model For An Extensible Editor

Inventors:

Ashraf Michail
Mark Alcazar
John Bedworth

ATTORNEY'S DOCKET NO. MS1-657US

TECHNICAL FIELD

The systems and methods described herein relate to extensible editors and, more particularly, the described implementations relate to event routing in extensible editors.

BACKGROUND

The process of editing electronic documents generally consists of processing events and key combinations received by an editor. From events and key combinations, an editing space is created. The editing space consists of a document state plus a view state (visual feedback). The view state includes a selection state (what is selected, what is shown as feedback), scroll position, etc. The event interacts with the current state, and an editing model is applied to manipulate feedback or to manipulate the document based on the feedback.

Extensible editors typically provide a set of document manipulation services that can enable macros to perform advanced tasks on the content of a document. With an extensible editor, developers can integrate extensions that define or re-define a manner in which the editor responds to events or key combinations and provides visual feedback to a user. For example, if a developer wishes to create a new look to a selection process or a highlighting process in an extensible editor, the developer can create an extension that receives input regarding cursor position, cursor movement, mouse actions, etc. The developer can design the extension to use this input to create visual feedback that differs from default selection services or highlight services of the editor. However, familiar behavior of the editor (such as clicking a "bold" button) is still retained by

the editor. In addition, an extension can also expose virtually any kind of functionality through its own interfaces.

Extensible editors are designed so that one or more extensions can be coupled with the editor. This is accomplished by implementing a set of interfaces to which the extensions must conform. The interfaces are utilized by the extensions to access a host of basic functions so that the extensions themselves are not required to implement such basic functions. The extensions instead utilize the basic functions to perform tasks that supplement or override functions performed by the editor.

When utilizing extensions in an extensible editor, however, a conflict problem can arise if multiple extensions are used simultaneously that act upon the occurrence of the same event or key combination. If a first extension reacts to an event in one way, but a second extension receives the event after the first extension has acted on it, the second extension may also act on the event and, as a result, override the action of the first extension. This problem can occur even when using only one extension, if the extension acts on an event or key combination but does not prevent the (default) editor from subsequently acting on the same event.

SUMMARY

An extensible editor for editing electronic documents and/or content is described herein. The extensible editor provides an event routing model that improves avoidance of extension-extension and editor-extension event handling conflicts. The extensible editor ("editor") includes three sets of interfaces for extension integration.

The first set of interfaces is part of a designer extensibility mechanism. The designer extensibility mechanism is used to couple an extension (also called a "designer") to the editor so the extension can utilize the event routing model of the editor. The designer extensibility mechanism provides the ability to connect an editor extension that can modify editing behavior. An attached designer receives events and key combinations in a predefined order and uses the set of interfaces to create custom editing extensions.

The designer extensibility mechanism includes an edit designer interface that has four methods: translate accelerator, pre-handle event, post-handle event and post-event notify. The methods act as callback routines whenever an event occurs in the editing environment of the editor. When an event comes into the editor, the four methods intercept the event at different points of the editor's event handling process. The editor invokes the methods sequentially. If multiple designers are utilized, the editor invokes the current method on each designer sequentially, in the order in which the designers were registered with the editor.

If a designer acts on an event and wants to prevent any subsequent designer (or the editor) from acting on the event, the designer "consumes" the event by returning an appropriate signal to the editor. The designer returns a different signal when subsequent processing is to continue on an event.

Key combinations entered through a keyboard (e.g., Ctrl-A, Alt-P, etc.) first pass through the "translate accelerator" method. This is done in the order that the designers were added to the editor. If an event is generated by keyboard input (i.e., a key combination was entered), then the designer acts upon the event and "consumes" the event so that subsequent designers or the default editor cannot subsequently act upon the event.

The post-editor event notify method is an exception to the rule that consumed events are not passed on to subsequent designers or the editor for further processing. This method is always called on all designers, regardless of whether, or when, an event is consumed. This allows each designer to clean up any internal states that may be anticipating an event that is consumed before reaching the designer.

The second set of interfaces is included in a selection services component of the editor. The selection services component provides designers (*i.e.*, editing extensions) with the ability to manage logical selections that are used by commands and other extensions, *i.e.*, the ability to modify the logical selection state of the editor. As a result, all editing commands and services will be able to interact with a custom selection model without having detailed knowledge of the designer that is implementing the selection.

The third set of interfaces is included in a highlight rendering services component. The highlight rendering component allows a user to modify the rendered character attributes of text without modifying the document content. This facility is critical for providing a mechanism for providing user feedback without affecting persistence, undo, etc.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of exemplary methods and arrangements of the present invention may be had by reference to the following detailed description when taken in conjunction with the accompanying drawings wherein:

Fig. 1 is an exemplary computer system on which the present invention may be implemented.

Fig. 2 is a block diagram of a computer having an extensible editor and several extensions coupled with the editor stored in memory.

Fig. 3 is a block diagram of an editor including a designer interface.

Fig. 4 is a block diagram of an event routing model utilized in an extensible editor.

Fig. 5 is a flow diagram of an event routing model for use in an extensible editor.

Fig. 6 is a block diagram of an editor including a selection services interface.

Fig. 7 is a block diagram of an editor including a highlighting services interface.

DETAILED DESCRIPTION

The invention is illustrated in the drawings as being implemented in a suitable computing environment. Although not required, the invention will be described in the general context of computer-executable instructions, such as program modules, to be executed by a computing device, such as a personal computer or a hand-held computer or electronic device. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including multi-processor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Exemplary Computer Environment

The various components and functionality described herein are implemented with a number of individual computers. Fig. 1 shows components of typical example of such a computer, referred by to reference numeral 100. The components shown in Fig. 1 are only examples, and are not intended to suggest any limitation as to the scope of the functionality of the invention; the invention is not necessarily dependent on the features shown in Fig. 1.

Generally, various different general purpose or special purpose computing system configurations can be used. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to, personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

The functionality of the computers is embodied in many cases by computer-executable instructions, such as program modules, that are executed by the computers. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Tasks might also be performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media.

The instructions and/or program modules are stored at different times in the various computer-readable media that are either part of the computer or that can be read by the computer. Programs are typically distributed, for example, on floppy disks, CD-ROMs, DVD, or some form of communication media such as a modulated signal. From there, they are installed or loaded into the secondary memory of a computer. At execution, they are loaded at least partially into the computer's primary electronic memory. The invention described herein includes these and other various types of computer-readable media when such media contain instructions programs, and/or modules for implementing the steps

described below in conjunction with a microprocessor or other data processors. The invention also includes the computer itself when programmed according to the methods and techniques described below.

For purposes of illustration, programs and other executable program components such as the operating system are illustrated herein as discrete blocks, although it is recognized that such programs and components reside at various times in different storage components of the computer, and are executed by the data processor(s) of the computer.

With reference to Fig. 1, the components of computer 100 may include, but are not limited to, a processing unit 120, a system memory 130, and a system bus 121 that couples various system components including the system memory to the processing unit 120. The system bus 121 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISAA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus also known as the Mezzanine bus.

Computer 100 typically includes a variety of computer-readable media. Computer-readable media can be any available media that can be accessed by computer 100 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer-readable media may comprise computer storage media and communication media. "Computer storage media" includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of

information such as computer-readable instructions, data structures, program modules, or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 110. Communication media typically embodies computer-readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more if its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media.

The system memory 130 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 131 and random access memory (RAM) 132. A basic input/output system 133 (BIOS), containing the basic routines that help to transfer information between elements within computer 100, such as during start-up, is typically stored in ROM 131. RAM 132 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 120. By way of example, and not limitation, Fig. 1 illustrates operating system 134, application programs 135, other program modules 136, and program data 137.

The computer 100 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, Fig. 1 illustrates a hard disk drive 141 that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive 151 that reads from or writes to a removable, nonvolatile magnetic disk 152, and an optical disk drive 155 that reads from or writes to a removable, nonvolatile optical disk 156 such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 141 is typically connected to the system bus 121 through an non-removable memory interface such as interface 140, and magnetic disk drive 151 and optical disk drive 155 are typically connected to the system bus 121 by a removable memory interface such as interface 150.

The drives and their associated computer storage media discussed above and illustrated in Fig. 1 provide storage of computer-readable instructions, data structures, program modules, and other data for computer 100. In Fig. 1, for example, hard disk drive 141 is illustrated as storing operating system 144, application programs 145, other program modules 146, and program data 147. Note that these components can either be the same as or different from operating system 134, application programs 135, other program modules 136, and program data 137. Operating system 144, application programs 145, other program modules 146, and program data 147 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and

information into the computer 100 through input devices such as a keyboard 162 and pointing device 161, commonly referred to as a mouse, trackball, or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 120 through a user input interface 160 that is coupled to the system bus, but may be connected by other interface and bus structures, such as a parallel port, game port, or a universal serial bus (USB). A monitor 191 or other type of display device is also connected to the system bus 121 via an interface, such as a video interface 190. In addition to the monitor, computers may also include other peripheral output devices such as speakers 197 and printer 196, which may be connected through an output peripheral interface 195.

The computer may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 180. The remote computer 180 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to computer 100, although only a memory storage device 181 has been illustrated in Fig. 1. The logical connections depicted in Fig. 1 include a local area network (LAN) 171 and a wide area network (WAN) 173, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and the Internet.

When used in a LAN networking environment, the computer 100 is connected to the LAN 171 through a network interface or adapter 170. When used in a WAN networking environment, the computer 100 typically includes a modem

172 or other means for establishing communications over the WAN 173, such as the Internet. The modem 172, which may be internal or external, may be connected to the system bus 121 via the user input interface 160, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 100, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, Fig. 1 illustrates remote application programs 185 as residing on memory device 181. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

Fig. 2 is a block diagram of a computer 200 having a processor 202 and memory 204. An extensible editor 206 stored in the memory 204 includes an event routing controller 208, a designer extensibility mechanism 210, a selection services component 212, and a highlight rendering services component 214. Three designers 216, 218, 220 are also stored in the memory 204. Each of the designers 216 - 220 communicates with the editor 206 via the designer extensibility mechanism 210. Each designer 216 – 220, as shown, also communicates with the selection services component 212 and the highlight rendering component 214. It is noted, however, that a designer 216 – 220 may communicate with only the selection services component 212 or the highlight rendering component 214 or with neither. However, as will become clear in the following discussion, each designer 216 – 220 must attach to the editor 206 through the designer extensibility mechanism 210.

A designer is an editor extension that is used to extend the functionality of the editor 206 and to customize the behavior of the editor 206. While the term "extension" is a generic term for a designer, the term "designer" is utilized in

several products of the MICROSOFT CORP., such as INTERNET EXPLORER 5.5, MSHTML (the HTML parsing and rendering engine of INTERNET EXPLORER that displays a document with editable content), the WebBrowser (ActiveX) control, etc. For purposes of the present discussion, terms specific to one or more of such products will be used. For example, in the present discussion, editor extensions will be referred to as designers. A reference to such a specific term (*e.g.*, designer) is intended to include reference to the more generic term (*e.g.*, extension).

The designers 216 - 220 work by intercepting events and commands occurring in, or received by, the editor 206. When one or more of the designers 216 - 220 intercepts an event, the designer can change how the editor 206 handles the event. Generally, a designer is written to either supplement or override the editor's behavior. Several designers may be attached to the editor 206 at once, thereby dynamically enabling multiple levels of custom functionality.

Designers offer a very powerful tool for customizing the editor 206. Virtually any part of the editor's behavior can be changed. For example, designers may be used to add spell checking capability to the editor 206, to add table editing functionality, to add annotation or revision-tracking capability, and so on. It is noted that, although only three designers 216 - 220 are shown in conjunction with the editor 206, any number of designers may be connected to the editor 206.

The designer extensibility mechanism 210, the selection services component 212, and the highlight rendering services 214 of the editor 206 shown in Fig. 2 provide specific functionality to the editor 206. Each of the modules and the functionality it provides will be discussed separately, in detail, below.

Designer Extensibility Mechanism

Fig. 3 is a block diagram of an editor 300 similar to the editor 206 shown in Fig. 2. The editor 300 includes a default event handler 301, an event routing controller 302 and a designer registry 303. The editor also includes an edit designer interface 304 that has several methods through which one or more designers (not shown) communicate with the editor 300. Each designer that is coupled with the editor 300 communicates with the editor 300 through the edit designer interface 304. Any coupled designer may then communicate with any other interfaces that are a part of the editor 300.

When a designer is added to the editor 300, the designer is registered in the designer registry 303. The event routing controller 302 accesses the designer registry 303 to determine the designers that are coupled with the editor 300. As will be discussed in greater detail below, the event routing controller 302 utilizes the designer registry 303 when routing events to attached designers.

The methods of the edit designer interface 304 are a translate accelerator method 306 (TranslateAccelerator), a pre-handle event method 308 (PreHandleEvent), a post-handle event method 310 (PostHandleEvent), and a post-editor event notify method 312 (PostEditorEventNotify). Each method 306 - 312 has two parameters, an event identifier 314 and an event object interface 316. In one implementation, the event identifier 314 is a value included in HTMLELEMENTEVENTS2 in mshtmdid.h, and the event object interface 316 is an IHTMLEventObj, which enables a designer to obtain more extended information about the event.

The methods 306 - 312 act as callback routines whenever an event occurs in the editing environment. In other words, whenever an event occurs that is

detected by the editor 300, each of the methods 306 - 312 is called, in a particular sequence, by the editor 300 to process the event. Processing the event may entail providing an external response to the event, responding internally to the event, not responding to the event, consuming the event and/or passing the event for further processing (*i.e.*, not consuming the event).

The event routing controller 302 determines when a particular method will be called. If multiple designers are registered with the editor 300 in the designer registry 303, the event routing controller 302 invokes the current method on each designer sequentially, in the order in which the designers were registered with the editor 300. Further explanation of the event routing technique will be explained below with continuing reference to the elements and reference numerals of Fig. 3.

Fig. 4 is a block diagram of an event routing model utilized in an extensible editor. In addition to the editor 300 shown in Fig. 3, Fig. 4 includes a first extension, Designer A 320, and a second extension, Designer B 330. Although only two designers 320, 330 are shown, it should be understood that virtually any number of designers may be added to the editor 300. The functionality of the routing mechanism when using more than two designers is similar to the description of the routing mechanism using the two designers 320, 330.

The editor 300 is designed to receive notification of an event 400. If the event 400 is a key combination input by a user, then the event routing controller 302 routes the event to the translate accelerator method 306. The event 400 is made available to a translation accelerator 306a in Designer A 320. The translation accelerator 306a of Designer A 320 may or may not provide a response to the event 400. If a response is provided to the event 400 by Designer A 320, then Designer A 320 may consume the event 400 to prevent Designer B 330 from

overriding the response to the event 400 by Designer A 320. To "consume" an event, a designer returns a value (S_OK) to the editor indicating that no further processing should be done on the event. If Designer A 320 does not respond to the event 400, then the event 400 will be made available to a translation accelerator 306b of Designer B 330. To indicate that an event should continue to be processed, a designer returns a different value to the editor (S_FALSE). Designer B 330 may then respond or not respond to the event 400 in the same way as described for Designer A 320. This process continues with any other designers that may be attached to the editor 300.

After each designer 320, 330 has had the opportunity to react to the event 400 (unless one of the designers 320, 330 has already consumed the event), control of the event 400 is returned to the event routing controller 302 of the editor 300. It is noted that the editor 300 may include its own translate accelerator that is designed to translate the key combination. This translate accelerator could be a part of the default event handler 301. In the preferred implementation, the default event handler 301 will receive the key combination (event 400) if not already consumed by one of the designers 320, 330. The default event handler 301 may or may not be configured to provide a response to the particular key combination. If it is configured to respond to a particular key combination, then an appropriate response is made; if not, then no response will be made to the key combination.

It is significant that if Designer B 330 is configured to act on a particular key combination, Designer B 330 may never receive that key combination if Designer A 320 consumes the combination. Therefore, it is noted that while the implementations described herein significantly improve conflict avoidance in extensible editors, when developing a new designer, careful consideration must be

given to the key combinations and events that will trigger actions by a designer. Also, multiple designers can be strategically registered with the editor 300 to avoid this situation.

If the event 400 is not a key combination, then the event routing controller makes the event 400 available to the designers 320, 330 via the pre-handle event method 308. Designer A 320 first has the opportunity to respond to the event 400. If Designer A 320 is configured to respond to the event 400, then it provides an appropriate response. After providing a response, Designer A 320 may either consume the event 400 or pass it along.

If, for example, Designer A 320 is an "auto correct" designer and Designer B 330 is a grammar checking designer, an event (entry of a word into the document) would be routed first through the auto correct designer to determine if the word should be corrected. After the word is checked (whether or not it is corrected), the grammar checking designer still requires notice of the event to perform its function. Therefore, Designer A 320 would act on the event but still make it available to Designer B 330

If the event 400 is not consumed by Designer A 320, then the event 400 is made available to Designer B 330. Designer B 330 has the same options of reacting to the event 400 as described above for Designer A 320. This is true for each subsequent designer attached to the editor 300.

After each designer has the opportunity to respond to the event 400 a first time, the event is passed to the default event handler 301 (assuming that the event 400 has not been previously consumed by a designer). The default event handler 301 then provides the default behavior of the editor 300 in response to the event 400. It is noted that if no designers are attached to the editor 300, then the editor

300 will simply provide the default editing behavior via the default event handler 301.

After the default event handler 301 has acted on the event 400, the designers 320, 330 are provided another opportunity to respond to the event 400. The event 400 is made available to the designers 320, 330 through the post-handle event method 310. The post-handle event processing is similar to the pre-handle event processing, occurring in the sequence in which the designers 320, 330 were registered with the editor 300.

By way of example, suppose that a developer wants to implement an "auto correct" designer that listens to key strokes. The designer, in this case, should receive an event after a typed character is inserted into the document (*i.e.*, PostHandleEvent) rather than before the character is inserted. Receiving the event after the default editor has inserted the character allows the designer to inspect the document with the correct content, allow undo of auto-correct behavior, etc.

After the post-handle event processing is concluded, the event 400 is processed by the default handler 301.

The post-editor event notify method 312 is called after the editor 300 has finished it post-handling of the event 400 or when a designer 320, 330 has consumed the event. In the case where the event 400 is consumed by a designer 320, 330 before a default action takes place and no post-handle event methods are called, a post-editor event notification module 312a, 312b is invoked to give the designers 320, 330 an opportunity to make a final response. For example, suppose a mouse down event starts a selection in a designer that implements basic text selection. If there is a mouse down event, the designer starts the selection. Now, if some other designer consumes the corresponding mouse up event, the designer

still needs to know about the mouse up event so it can terminate the selection and stop responding to mouse move events.

Fig. 5 is a flow diagram of the preferred implementation of the event routing model described above for use in an extensible editor 300. Continuing reference will be made to the elements and reference numerals of Fig. 2 – Fig. 4 in the description of the flow diagram.

At step 500, an event 400 is received by the editor 300. The event routing controller 302 determines if the event 400 is a key combination at step 502. If the event is a key combination ("Yes" branch, step 502), then the key combination is translated at step 504 by the designers (if they are configured to translate the particular command) or by the default event handler 301 if no designer translates the key combination and the editor 300 is configured to do so.

If the event is not a key combination ("No" branch, step 502) the editor 300 determines if a designer 320, 330 is attached to the editor 300 at step 508. If there is not an attached designer ("No" branch, step 508), the event is processed by the default event handler 301 at step 510. If, however, a designer 320, 330 is attached to the editor 300 ("Yes" branch, step 508), then the event 400 is made available to Designer A 320 for pre-handling at step 512. After Designer A 320 has had the opportunity to act on the event 400, Designer B 330 ("Yes" branch, step 514) has the event 400 made available for processing at step 512. There is no other designer attached ("No" branch, step 514), so the event (if not previously consumed), is passed to the default event handler 301 and processed at step 516.

After the event 400 is processed by the default event handler 301 of the editor 300, the event is made available to Designer A 320 for processing at step 518. Since there is another designer (Designer B 330) ("Yes" branch, step 520),

the event 400 is made available to Designer B 330 for processing at step 518. When there are no more designers to process the event 400 ("No" branch, step 520), the event is processed by the default event handler 301 at step 522.

It is noted that the above discussion assumes that the event 400 is not consumed by the default event handler 301 or the designers 320, 330 and is processed by each method of the editor 300 and the designers 320, 330. Once an event is consumed, further processing of the event terminates.

After the event has been through the pre-handle event method 308 and the post-handle event method 310, the post-editor event notification method 312 is called (step 524). Unless consumed by the default event handler 310 or one of the designers 320, 330, each designer 320, 330 is notified of any response to the event 400 from any other module as previously described.

Designer Extensibility Mechanism Interfaces

In addition to the edit designer interface 304 described above, the designer extensibility mechanism 210 includes an edit services interface. In MSHTML, the edit services interface - designated as IHTMLEditServices - is used to add or remove edit designers and control the current selection. Although the general descriptions of the designer extensibility mechanism can be applied and implemented in any extensible editor and extensions therefor, for discussion purposes, the described implementation will refer to MICROSOFT MSHTML terminology to describe interfaces exposed by a designer extensibility mechanism to allow extensions to be properly coupled to communicate with an extensible editor. Those skilled in the art will appreciate the functions enabled by the described interfaces to implement custom extensions for the extensible editor.

The following are detailed descriptions of the edit designer interface and the edit services interface.

Edit Designer Interface (IHTMLEditDesigner)

The edit designer interface includes the following methods:

TranslateAccelerator

Description: Called by MSHTML to translate a key combination entered by a user into an appropriate command.

Syntax:

HRESULT TranslateAccelerator (

DISPID *inEvtDispId*,

IHTMLEventObj **pIEventObj*

Parameters:

inEvtDispId

[in] DISPID that specifies the event.

pIEventObj

[in] Pointer to an IHTMLEventObj interface that specifies the event.

Return Values:

Returns S_OK to indicate that the event has been completely handled and that no further processing should take place, either by other edit designers or the MSHTML Editor. Returns S_FALSE to indicate that other edit designers and the MSHTML Editor should perform their processing the this event.

PreHandleEvent

Description: Called by MSHTML before the MSHTML Editor processes an event, so that the designer can provide its own event handling behavior.

Syntax:

HRESULT PreHandleEvent (

DISPID *inEvtDispId*,

IHTMLEventObj **pIEventObj*

Parameters:

inEvtDispId

[in] DISPID that specifies the event.

pIEventObj

[in] Pointer to an IHTMLEventObj interface that specifies the event.

Return Values:

Returns S_OK to indicate that the event has been completely handled and that no further processing should take place, either by other edit designers or the MSHTML Editor. Returns S_FALSE to indicate that other edit designers and the MSHTML Editor should perform their pre-event processing.

PostHandleEvent

Description: Called by MSHTML after the MSHTML Editor processes an event, so that the designer can provide its own event handling behavior.

Syntax:

HRESULT PostHandleEvent (

DISPID *inEvtDispId*,

IHTMLEventObj **pIEventObj*

Parameters:

inEvtDispId

[in] DISPID that specifies the event.

pIEventObj

[in] Pointer to an IHTMLEventObj interface that specifies the event.

Return Values:

Returns S_OK to indicate that the event has been compoetley handled and that no further processing should take place, either by other edit designers or the MSHTML Editor. Returns S_FALSE to indicate that other edit designers and the MSHTML Editor should perform their post-event processing.

PostEditorEventNotify

Description: Called by MSHTML after an event has been handled by the MSHTML Editor and any registered edit designers.

Syntax:

HRESULT PostHandleEvent (

DISPID *inEvtDispId*,

IHTMLEventObj **pIEventObj*

Parameters:

inEvtDispId

[in] DISPID that specifies the event.

pIEventObj

[in] Pointer to an IHTMLEventObj interface that specifies the event.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Edit Services Interface (IHTMLEditServices)

The edit designer interface includes the following methods:

AddDesigner

Description: Registers an IHTMLEditDesigner interface to receive event notification from the editor.

Syntax:

```
HRESULT AddDesigner (
    IHTMLEditDesigner *pIDesigner
);
```

Parameters:

pIDesigner

[in] Pointer to an IHTMLEditDesigner interface to register for event notification.

Return Values:

Returns S_OK if successful, or an error value otherwise.

GetSelectionServices

Description: Registers an IHTMLEditDesigner interface to receive event notification from the editor.

Syntax:

```
HRESULT GetSelectionServices (
    IMarkupContainer *pIContainer
```

ISelectionServices **ppSelSvc*

);

Parameters:

**pIContainer*

[in] Pointer to an IMarkupContainer interface for which an ISelectionServices interface is desired.

***ppSelSvc*

[out] Address of a pointer to a variable that receives an ISelectionServices interface pointer for the ISelectionServices interface on the editor's selection object.

Return Values:

Returns S_OK if successful, or an error value otherwise.

MoveToSelectionAnchor

Description: Moves a markup pointer to the location of an anchor for the current selection.

Syntax:

HRESULT MoveSelectionToAnchor (

IMarkupPointer **pIStartAnchorr*

);

Parameters:

**pIStartAnchor*

[in] Pointer to an IMarkupPointer interface to move the location of an anchor for the selection.

Return Values:

Returns S_OK if successful, or an error value otherwise.

MoveToSelectionEnd

Description: Moves markup pointer to the end of the current selection.

Syntax:

```
HRESULT MoveToSelectionEnd (
        IMarkupPointer *pIEndAnchor
);
```

Parameters:

**pIEndAnchor*

[in] Pointer to an IMarkupPointer interface to move to the end of the current session.

Return Values:

Returns S_OK if successful, or an error value otherwise.

RemoveDesigner

Description: Unregisters a designer from the editor.

Syntax:

```
HRESULT RemoveDesigner (
        IHTMLEditDesigner *pIDesigner
);
```

Parameters:

pIDesigner

[in] Pointer to the IHTMLEditDesigner interface to remove from the event notification queue.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Selection Services

Selection services provides extensions a way to modify a selection process of an extensible editor to which the designers are coupled. Although the general descriptions of the selection services can be applied and implemented in any extensible editor and extensions therefor, for discussion purposes, the described implementation will refer to MICROSOFT MSHTML terminology to describe interfaces exposed by a selection services component to allow extensions to properly communicate with an extensible editor to utilize the selection services component. Those skilled in the art will appreciate the functions enabled by the described interfaces to implement custom extensions for the extensible editor.

Fig. 6 is a block diagram of an extensible editor 600 that includes a designer interface 602, an event routing mechanism 604, and a selection services component 606. The selection services component 606 includes several interfaces: a selection services interface 608 (ISelectionServices), a selection services listener interface 610 (ISelectionServiceListener), an element segment interface 612 (IElementSegment), a segment list interface 614 (ISegmentList), and a segment interface 616 (ISegment). These interfaces 608 – 616 will be discussed in greater detail, below.

The role of the selection services interfaces 608 – 616 is to provide designers or other editing extensions with the ability to modify the logical selection state. Consequently, all editing commands and services can interact with a custom selection model without having detailed knowledge of the designer that is implementing the selection.

For example, the "bold" command is able to implement the operation of making something bold without having any knowledge of the specifics of a given

designer. The command is only aware of what part of the document is selected, and it is configured to make the selected region of the document bold. A user still only has to know the click on a "bold" icon to invoke the "bold" command.

Selection Service Interface (ISelectionServices)

The selection services interface 608 provides methods to programmatically clear, add and remove segments from a selection object. The methods include an add element segment method 618 (AddElementSegment), a get markup container method 620 (GetMarkupContainer), a get selection services listener method 622 (GetSelectionServicesListener), an add segment method 624 (AddSegment), a remove segment method 626 (RemoveSegment), and a set selection type method 628 (SetSelectionType). The following are detailed description of the available selection services interface 608 methods.

AddElementSegment

Description: The add element segment method 618 creates an IElementSegment interface for an element in a markup container and adds the segment to the editable selection.

Syntax:

```
HRESULT AddElementSegment (
        IHTMLElement *pIElement,
        IElementSegment **ppISegmentAdded
);
```

Parameters:

**pIElement*

[in] Pointer to an IHTMLElement interface that specifies the element to add to the selection.

**ppISegmentAdded*

[out] Address of a pointer to a variable that receives an IElementSegment interface pointer for the element segment added to the selection environment.

Return Values:

Returns S_OK if successful, or an error value otherwise.

GetMarkupContainer

Description: The get markup container method 620 retrieves the markup container for the current editable selection.

Syntax:

```
HRESULT GetMarkupContainer (
    IMarkupContainer **ppIContainer
);
```

Parameters:

**ppIContainer*

[out] Address of a pointer to a variable that receives an IMarkupContainer interface pointer to the interface for the markup container that contains the current editable selection.

Return Values:

Returns S_OK if successful, or an error value otherwise.

GetSelectionServicesListener

Description: The get selection services listener 622 method retrieves an ISelectionServicesListener interface for the current editable selection so that the editor can process certain selection events.

Syntax:

HRESULT GetSelectionServicesListener (

ISelectionServicesListener ***ppISelectionServicesListener*

);

Parameters:

***ppISelectionServicesListener*

[out] Address of a pointer to a variable that receives an ISelectionServicesListener interface pointer to the interface that the editor will use with the current editable selection.

Return Values:

Returns S_OK if successful, or an error value otherwise.

AddSegment

Description: The add segment method 624 creates an ISegment interface for the content between two markup pointers in a markup container, and adds the segment to the editable selection.

Syntax:

HRETURN AddSegment (

IMarkupPointer **pIStart*,

IMarkupPointer **pIEnd*,

ISegment ***ppISegmentAdded*

);

Parameters:

**pIStart*

[in] Pointer to an IMarkupPointer interface that specifies the start point for adding the segment.

**pIEnd*

[in] Pointer to an IMarkupPointer interface that specifies the end point for adding the segment.

***ppISegmentAdded*

[out] Address of a pointer to a variable that receives an ISegment interface pointer to the interface for the added segment in the selection environment.

Return Values:

Returns S_OK if successful, or an error value otherwise.

RemoveSegment

Description: The remove segment method 626 (RemoveSegment) removes a segment from the editable selection.

Syntax:

```
HRESULT RemoveSegment (
    ISegment *pISegment
);
```

Parameters:

**pISegment*

[in] Pointer to an ISegment interface that specifies the segment to remove.

Return Values:

Returns S_OK if successful, or an error value otherwise.

SetSelectionType

Description: The set selection type method 628 (SetSelectionType) sets the selection type and clears any existing selection.

Syntax:

HRESULT SetSelectionType;

SELECTION_TYPE *eType*

ISelectionServicesListener **pIListener*

);

Parameters:

eType

[in] SELECTION_TYPE enumeration that specifies the type of selection to set.

**pIListener*

[in] Optional. Pointer to an ISelectionServiceListener interface specifying the interface to associate with this selection. Set to NULL if unused. NOTE: Although this parameter is optional, without this parameter, the editor will not be able to restore the selection.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Element Segment Interface (IElementSegment)

The element segment interface 610 provides methods that control a fragment of HTML markup in the current editable selections that consists of a single element. The element segment interface 612 includes a get element method 630 (GetElement), an 'is primary' method 632 (IsPrimary), and a set primary method 634 (SetPrimary). To obtain an IElementSegment interface for a fragment

of HTML markup representing an element, the ISelectionServices::AddElementSegment method is used.

The selection object uses element segments to mark fragments of HTML markup that are whole elements, in particular control elements.

GetElement

Description: The get element method 612 (GetElement) retrieves the element to which this segment refers.

Syntax:

```
HRESULT GetElement (
        IHTMLElement **ppIElement
);
```

Parameters:

***ppIElement*

[out] Address of a pointer to a variable that receives an IHTMLElement interface pointer for the interface representing the element to which the segment refers.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Is Primary

Description: The is primary method 632 (IsPrimary) determines whether the control element represented by this segment is the primary element of a multi-element selection. The primary element of a multiple selection is typically the first one chosen by a user when a selection was made. The primary element typically has distinctive handles that indicate it is the primary element. For

example, the primary element might have white handles while the other elements have black ones).

Syntax:

HRESULT IsPrimary (

BOOL **pfPrimary*

);

Parameters:

**pfPrimary*

[out] Pointer to a BOOL that receives TRUE if the element is the primary element, or FALSE otherwise.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Set Primary

Description: The set primary method (SetPrimary) sets or unsets a control element as a primary element in a control selection. The primary element of a multiple selection is typically the first one chosen by a user when a selection was made. The primary element typically has distinctive handles that indicate it is the primary element. For example, the primary element might have white handles while the other elements have black ones.

Syntax:

HRESULT SetPrimary

BOOL *fPrimary*

);

Parameters:

fPrimary

[in] BOOL that specifies TRUE to set the element as the primary element, or FALSE to unset it as primary.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Segment Interface (ISegment)

The segment interface 616 provides a method that creates containers (segments) for fragments of HTML markup in the current editable selection. These segments can include both a range of elements and element fragments. The segment interface 616 includes a get pointers method 636 (GetPointers).

GetPoiners

Description: The get pointers method 636 (GetPointers) positions markup pointers at the start and end of the selection segment.

Syntax:

```
HRESULT GetPointers (
        IMarkupPointer *pIStart,
        IMarkupPointer *pIEnd
);
```

Parameters:

pIStart

[in] Pointer to an IMarkupPointer interface that specifies the markup pointer to position at the beginning of the segment.

pIEnd

[in] Pointer to an IMarkupPointer interface that specifies the markup pointer to position at the end of the segment.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Segment List Interface (ISegmentList)

The segment list interface 614 provides methods that access information about a list of the segments in the current selection. The segment list interface 614 includes a create iterator method 638 (CreateIterator), a get type method 640 (GetType), and an 'is empty' method 640 (IsEmpty).

CreateIterator

Description: The create iterator method 638 creates an ISegmentListIterator interface used for traversing the members of a segment list.

Syntax:

```
HRESULT CreateIterator (
    ISegmentListIterator **ppIIter
);
```

Parameters:

ppIIter

[out] Address of a pointer to a variable that receives an ISegmentListIterator interface pointer for the newly created ISegmentListIterator.

Return Values:

Returns S_OK if successful, or an error value otherwise.

GetType

Description: The get type method 640 retrieves the type of the selection.

Syntax:

HRESULT GetType (

SELECTION_TYPE *peType

);

Parameters:

peType

[out] Pointer to a variable of type SELECTION_TYPE that receives the selection type value.

Return Values:

Returns S_OK if successful, or an error value otherwise.

IsEmpty

Description: The 'is empty' method 642 determines whether the segment list is empty.

Syntax:

HRESULT IsEmpty (

BOOL *pfEmpty

) ;

Parameters:

pfEmpty

[out] Pointer to a variable of type BOOL that receives TRUE if the segment list is empty, or FALSE if it is not empty.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Selection Services Listener Interface (ISelectionServicesListener)

The selection services listener interface 610 provides methods that the editing component of MSHTML calls whenever certain events fire for a selection object that has a registered ISelectionServicesListener interface. This interface provides processing for undo events, for selection type changes, and whenever the mouse pointer exits the scope of an element in the editable selection. An application should supply an implementation of this interface for a selection object so that the editing component of MSHTML can respond to these events. The selection services listener interface 610 includes a begin selection undo method (BeginSelectionUndo) 644, an end selection undo method 646 (EndSelectionUndo), a get type detail method 648 (GetTypeDetail), an 'on change type' method 650 (OnChangeType), and an 'on selected element exit' method 652 (OnSelectedElementExit). To register an ISelectionServicesListener interface for a particular selection object, the ISelectionServices::SetSelectionType method or ISelectionServices::OnChangeType method is used.

BeginSelectionUndo

Description: The begin selection undo method 644 is called by the editor 600 when an editing operation is beginning that may result in a change in selection after the editing operation. This method exists so that the designers may place their own units on an Undo queue so that a selection may be restored to its original state when the editing process was started.

Syntax:

HRESULT BeginSelectionUndo (VOID);

Parameters:

None.

Return Values:

Returns S_OK if successful, or an error value otherwise.

EndSelectionUndo

Description: The end selection undo method 644 is called by the editor 600 at the end of an editing operation that may result in a change in selection after the editing operation. This method exists so that the designers may place their own units on an Undo Queue so that a selection may be restored to its original state when the editing process was started.

Syntax:

HRESULT EndSelectionUndo (VOID);

Parameters:

None.

Return Values:

Returns S_OK if successful, or an error value otherwise.

GetTypeDetail

Description: The get type detail method 648 is called by MSHTML to obtain the name of the selection type. This method allows a host application to provide the name of a selection type when implementing a custom selection mechanism. MSHTML will return a value of 'undefined' if the host does not implement this method.

Syntax:

```
HRESULT GetTypeDetail (
    BSTR *pTypeDetail
);
```

Parameters:

pTypeDetail

[out] BSTR that specifies the name of the selection type.

Return Values:

Returns S_OK if successful, or an error value otherwise.

OnChangeType

Description: The 'on change type' method 650 is called by the editor 600 when the type of a selection changes. This method is used to implement custom processing that should take place when a selection is initiated or when a selection changes type.

Syntax:

HRESULT OnChangeType (

SELECTION_TYPE *eType*,

ISelectionServicesListerner **pIListener*

);

Parameters:

eType

[in] SELECTION_TYPE enumeration that specified the new selection type.

pIListener

[in] Optional. Pointer to an ISelectionServicesListener interface to register with the new selection. Can be set to NULL.

Return Values:

Returns S_OK if successful, or an error value otherwise.

OnSelectedElementExit

Description: The 'on selected element' exit method 652 is called by the editor 600 whenever an element that intersects selection undo is removed from the document. This method exists so that the selection can be updated by the extensible editor (either removed or adjusted).

Syntax:

```
HRESULT OnSelectedElementExit (
        IMarkupPointer *pIElementStart,
        IMarkupPointer *pIElementEnd,
        IMarkupPointer *pIElementContentStart,
        IMarkupPointer *pIElementContentEnd
);
```

Parameters:

**pIElementStart*

[in] Pointer to an IMarkupPointer interface specifying the point just before the element's opening tag.

**pIElementEnd*

[in] Pointer to an IMarkupPointer interface specifying the point just after the element's closing tag.

**pIElementContentStart*

[in] Pointer to an IMarkupPointer interface specifying the point just after the element's opening tag.

**pIElementContentEnd*

[in] Pointer to an IMarkupPointer interface specifying the point just before the element's closing tag.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Highlight Rendering Services

Highlight rendering services allows a user to modify the rendered character attributes of text without modifying the document content. This facility is critical for providing a mechanism for providing user feedback without modifying the document content. This component is critical for providing a mechanism for providing user feedback without affecting persistence, undo, etc.

Fig. 7 is a block diagram of an extensible editor 700 that includes a designer interface 702, an event routing mechanism 704, and a highlight rendering services component 706. The highlight rendering services component 706 includes two interfaces: a highlight services interface 708 (IHighlightRenderingServices), and a highlight segment interface 710 (IHighlightSegment). These interfaces 708, 710 will be discussed in greater detail, below.

Highlight Rendering Services Interface (IHighlightRenderingServices)

The highlight rendering services interface 708 provides methods that enable a designer to control which sections of a document are highlighted on the screen and the style of highlighting. The methods include an add segment method 712 (AddSegment), a move segment to pointers method 714 (MoveSegmentToPointers), and a remove segment method 716 (RemoveSegment). The following are detailed description of the available selection services interface 608 methods.

AddSegment

Description: The add segment method 712 creates a highlight segment for the markup between two display pointers and highlights it according to a specified rendering style.

Syntax:

```
HRESULT AddSegment (
        IDisplayPointer *pDispPointerStart,
        IDisplayPointer *pDispPointerEnd,
        IHTMLRenderStyle *pIRenderStyle,
        IHighlightSegment **ppISegment
);
```

Parameters:

pDispPointerStart

[in] Pointer to an IDisplayPointer interface representing the start point of the segment to be highlighted.

pDispPointerEnd

[in] Pointer to an IDisplayPointer interface representing the end point of the segment to be highlighted.

pIRenderStyle

[in] Pointer to an IHTMLRenderStyle interface representing the style with which to render the specified segment.

ppISegment

[out] Address of a pointer to a variable that receives an IHighlightSegment interface pointer for the interface that

represents the highlight segment between pDispPointerStart and pDispPointerEnd.

Return Values:

Returns S_OK if successful, or an error value otherwise.

MoveSegmentToPointers

Description: The move segments to pointers method 714 redefines a highlight segment and its style.

Syntax:

```
HRESULT MoveSegmentToPointers (
        IHighlightSegment *pISegment,
        IDisplayPointer *pDispPointerStart,
        IDisplayPointer *pDispPointerEnd
);
```

Parameters:

pISegment

[in] Pointer to an IHighlightSegment interface to redefine.

pIDispPointerStart

p[in] Pointer to an IDisplayPointer interface for the new start point of the highlight segment.

pIDispPointerEnd

[in] Pointer to an IDisplayPointer interface for a new endpoint of the highlight segment.

Return Values:

Returns S_OK if successful, or an error value otherwise.

RemoveSegment

Description: The remove segment method 716 removes a highlight segment from a collection of segments that are highlighted.

Syntax:

```
HRESULT RemoveSegment (
    IHighlightSegment *pISegment
);
```

Parameters:

pISegment

[in] Pointer to an IHighlightSegment interface to remove.

Return Values:

Returns S_OK if successful, or an error value otherwise.

Highlight Segment Interface (IHighlightSegment)

The highlight segment interface 710 enables a user to control a highlighed section of a document. This interface does not provide any methods of its own beyond those available from it parent interface, ISegment.

Description: The highlight segment interface 710 provides type checking for the segments added or moved from the highlighted sections through the IHighlightRenderingServices interface.

Remarks: This interface does not provide any methods of its own beyond those available from its parent interface, ISegment.

Conclusion

The services described above provide an applications program interface (API) for an extensible editor (MSHTML). The interfaces and the methods associated with each interface are summarized as follows:

IHTMLEditServices
- AddDesigner
- GetSelectionServices
- MoveToSelectionAnchor
- MoveToSelectionEnd
- RemoveDesigner

IHTMLEditDesigner
- TranslateAccelerator
- PreHandleEvent
- PostHandleEvent
- PostEditorEventNotify

ISelectionServices
- AddElementSegment
- GetMarkupContainer
- GetSelectionServicesListener
- AddSegment
- RemoveSegment
- SetSelectionType

ISelectionServicesListener
- BeginSelectionUndo
- EndSelectionUndo
- GetTypeDetail
- OnChangeType
- OnSelectedElementExit

ISegmentList
- CreateIterator
- GetType
- IsEmpty

ISegment

IElementSegment
- GetElement
- IsPrimary
- SetPrimary

IHighlightRenderingServices
- AddSegment
- MoveSegmentToPointers
- RemoveSegment

IHighlightSegment

The interfaces can be utilized by an extension coupled with the extensible editor to add new features to the editor, to augment existing features, or to override the editor's default behavior. Extensions can be used to modify the editor to provide customized feedback and to present a rich editing experience to a user.

Although details of specific implementations and embodiments are described above, such details are intended to satisfy statutory disclosure obligations rather than to limit the scope of the following claims. Thus, the invention as defined by the claims is not limited to the specific features described above. Rather, the invention is claimed in any of its forms or modifications that fall within the proper scope of the appended claims, appropriately interpreted in accordance with the doctrine of equivalents.

CLAIMS

1. An electronic document editor, comprising:

a default event handler to process editing events;

a designer extensibility mechanism to communicate with an extension coupled with the editor, the extension being configured to process at least one of the editing events; and

wherein the designer extensibility mechanism provides the editing events to the extension prior to the default event handler processing the editing events.

2. The electronic document editor as recited in claim 1, further comprising an extension interface having a pre-handle event method through which the designer extensibility mechanism provides the events to the extension.

3. The electronic document editor as recited in claim 1, wherein the designer extensibility mechanism further provides the editing events to the extension after the default event handler processes the editing events.

4. The electronic document editor as recited in claim 3, further comprising an extension interface having a pre-handle event method and a post-handle event method, the designer extensibility mechanism providing the editing events to the extension through the pre-handle event method prior to the default event handler processing the editing events, and providing the editing events to the extension through the post-handle event method after the default event handler has processed the editing events.

5. The electronic document editor as recited in claim 1, wherein:

the extension is a first extension; and

the designer extensibility mechanism further provides the editing events to a second extension after the editing events have been provided to the first extension, and prior to the default event handler processing the editing events.

6. The electronic document editor as recited in claim 5, wherein the designer extensibility mechanism further provides the editing events to the first extension and the second extension after the default event handler processes the editing events.

7. The electronic document editor as recited in claim 1, wherein:

the extension is a first extension;

the designer extensibility mechanism further provides the editing events to a second extension; and

the designer extensibility mechanism further provides notice to the first extension of any action taken on an event by the second extension or the default event handler.

8. In an extensible editor having an editor extension coupled therewith, the editor having an edit designer interface comprising a pre-handle event method, the pre-handle event method comprising:

routing an editing event to the editor extension before the editor acts on the editing event; and

receiving notification from the editor extension indicating whether the editor should continue to process the editing event after the editing event has been routed to the editor extension.

9. The method as recited in claim 8, wherein the routing an editing event to the editor extension further comprises routing an event identifier to the editor extension, the event identifier uniquely identifying an editing event.

10. The method as recited in claim 8, wherein the routing an editing event to the editor extension further comprises routing an event object interface to the editor extension, the event object interface providing the editor extension with means to process the editing event.

11. In an extensible editor having an editor extension coupled therewith, the editor having an edit designer interface comprising a post-handle event method, the post-handle event method comprising:

routing an editing event to the editor extension after the editor acts on the editing event; and

receiving notification from the editor extension indicating whether the editor should continue to process the editing event after the editing event has been routed to the editor extension.

12. The method as recited in claim 11, wherein the routing an editing event to the editor extension further comprises routing an event identifier to the editor extension, the event identifier uniquely identifying an editing event.

13. The method as recited in claim 11, wherein the routing an editing event to the editor extension further comprises routing an event object interface to the editor extension, the event object interface providing the editor extension with means to process the editing event.

14. A system, comprising:

an extensible editor that processes editing events, the extensible editor having an event routing controller and a default event handler;

an extension coupled with the extensible editor for processing the editing events; and

wherein the event routing controller provides an editing event received by the editor to the extension prior to providing the editing event to be processed by the default event handler.

15. The system as recited in claim 14, wherein:

the extension is a first extension;

the system further comprises a second extension for processing the editing events; and

the event routing controller provides the editing event to the second extension prior to providing the event to the default event handler.

16. The system as recited in claim 14, wherein the event routing controller provides the editing event to the extension after the default event handler has processed the editing event.

17. The system as recited in claim 14, wherein:

the extension is a first extension;

the system further comprises a second extension for processing the editing events;

the event routing controller routes editing events to the first extension, the second extension and the default event handler; and

each event is notified of any action taken in response to the editing event by the other extension or by the default event handler.

18. The system as recited in claim 14, wherein the event routing controller further provides an event identifier uniquely identifying the editing event.

19. The system as recited in claim 14, wherein the event routing controller provides an event object interface to the extension to allow the extension to access information regarding the editing event.

20. The system as recited in claim 14, wherein the editor further comprises an edit designer interface that includes a pre-handle event method for providing the event to the extension prior to the default event handler receiving the event.

21. The system as recited in claim 14, wherein:

the editor further comprises an edit designer interface that includes a post-handle event method for providing the editing event to the extension after the default event handler has processed the event; and

the event routing controller is further configured to provide the editing event to the extension through the edit designer interface.

22. The system as recited in claim 14, wherein:

the extension is a first extension;

the editor further comprises an edit designer interface that includes a post-editor event notify method that is called by the event routing controller after the editing event has been processed by the first extension, a second extension and the default event handler to provide data to the first extension and the second extension regarding actions taken in response the editing event.

23. A designer attached to an editor, comprising a pre-event handler that processes an editing event from the editor before the editor processes the event.

24. The designer as recited in claim 23, further comprising a post-event handler that processes the editing event after the editor processes the editing event.

25. The designer as recited in claim 23, wherein the pre-event handler processes the editing event and notifies the editor to prevent further processing on the event.

26. The designer as recited in claim 23, wherein the default event handler responds to the editing event by notifying the editor to continue processing the event.

27. The designer as recited in claim 23, further comprising a post-event handler that processes the editing event after the editor processes the editing event, and notifies the editor to prevent further processing on the event.

28. The designer as recited in claim 23, further comprising a post-event handler that processes the editing event after the editor processes the event, and notifies the editor to continue processing the editing event.

29. An editor that communicates with a first designer and a second designer, comprising:

a default event handler; and

an edit designer interface that includes a pre-handle event method to process an event before the default event handler processes the event, and a post-handle event method to process the event after the default event handler has processed the event.

30. The editor as recited in claim 29, further comprising a post-editor event notify method that is called to notify the first extension of an action taken by the second extension when processing the event.

31. The designer as recited in claim 23, further comprising a translate accelerator method that translates commands received by the editor.

32. The designer as recited in claim 23, wherein the pre-handle event method and the post-handle event method include an event ID parameter that uniquely identifies the event.

33. The designer as recited in claim 23, wherein the pre-handle event method and the post-handle event method include an event object interface that allows the extensions to obtain information about the event.

34. An edit designer interface in an extensible editor, comprising:

a pre-handle event method used to send an editing event to a designer attached to the editor prior to the editor processing the editing event; and

a post-handle event method used to send the editing event to the designer after the editor has processed the editing event.

35. The edit designer interface as recited in claim 34, wherein:

the designer is a first designer;

the edit designer interface further comprises a post-editor event notify method used to notify a second designer attached to the editor of an action taken by the first designer when the first designer processed the editing event.

36. The edit designer interface as recited in claim 34, wherein the pre-handle event method and the post-handle event method include an event ID parameter that uniquely identifies the editing event.

37. The edit designer interface as recited in claim 34, wherein the pre-handle event method and the post-handle event method include an event object interface associated with the editing event through which the designer can obtain information regarding the editing event.

38. A method for processing events in an extensible editor that communicates with a first extension and a second extension, the method comprising:

sending an event to the first extension; and

receiving a signal from the first extension indicating whether to continue processing the event.

39. The method as recited in claim 38, further comprising notifying the second extension about actions taken by the first extension in response to receiving the event if the signal indicates that the event should not be processed further.

40. The method as recited in claim 38, further comprising processing the event if the signal indicates that processing should continue.

41. The method as recited in claim 38, further comprising sending the event to the second extension if the signal indicates that processing should continue.

42. The method as recited in claim 38, further comprising:

determining if the event is a command; and

if the event is a command, translating the command and withholding the event from the extensions.

43. The method as recited in claim 38, further comprising:

determining if the event is a command; and

if the event is a command, translating the command and notifying the extensions that the command has been processed.

44. A computer-readable medium having computer-executable instructions that, when executed on a computer, perform the following steps:

detect an editing event;

routing the editing event to a designer;

receiving a response from the designer that indicates whether the editing event was consumed by the first designer.

45. The computer-readable medium as recited in claim 44, further comprising computer-executable instructions to perform the following step:

processing the editing event if the response from the designer indicates that the editing event was not consumed by the first designer.

46. The computer-readable medium as recited in claim 44, wherein the designer is a first designer, and further comprising computer-executable instructions to perform the following step:

notifying a second designer that the first designer consumed the event if the response from the first designer indicates that the first designer consumed the event.

47. The computer-readable medium as recited in claim 44, wherein the designer is a first designer, and further comprising computer-executable instructions to perform the following step:

routing the event to the second designer if the response from the first designer indicates that the first designer did not consume the event.

ABSTRACT

An extensible editor allows integration of extensions that modify the editor's default behavior and provide customized feedback to users. The editor includes an event routing model that works to decrease the occurrence of conflicts between the editor and extensions and between extensions. Upon the occurrence of an event, the editor routes the event to each extension before the editor's default handling of the event occurs. When an extension responds to an event, the extension may "consume" the event by indicating to the editor not to allow further processing of the event. After an event has been pre-processed by each extension, the default editor acts on the event. The editor then routes the event to each extension again, to allow each extension to process the event after the default editor has acted. When the post-processing is completed, each extension is notified of the actions taken by the editor and by each of the other extensions.

The editor includes interfaces through which extensions are connected to the editor and through which selection services and highlight rendering services are provided. The selection services interfaces provide a clear separation of a logical selection position in the document and the visual feedback provided for the selection, allowing extensions to be designed that provide customized selection feedback. The highlight rendering services interfaces provide an extension with the ability to augment an existing selection without modifying the actual document.

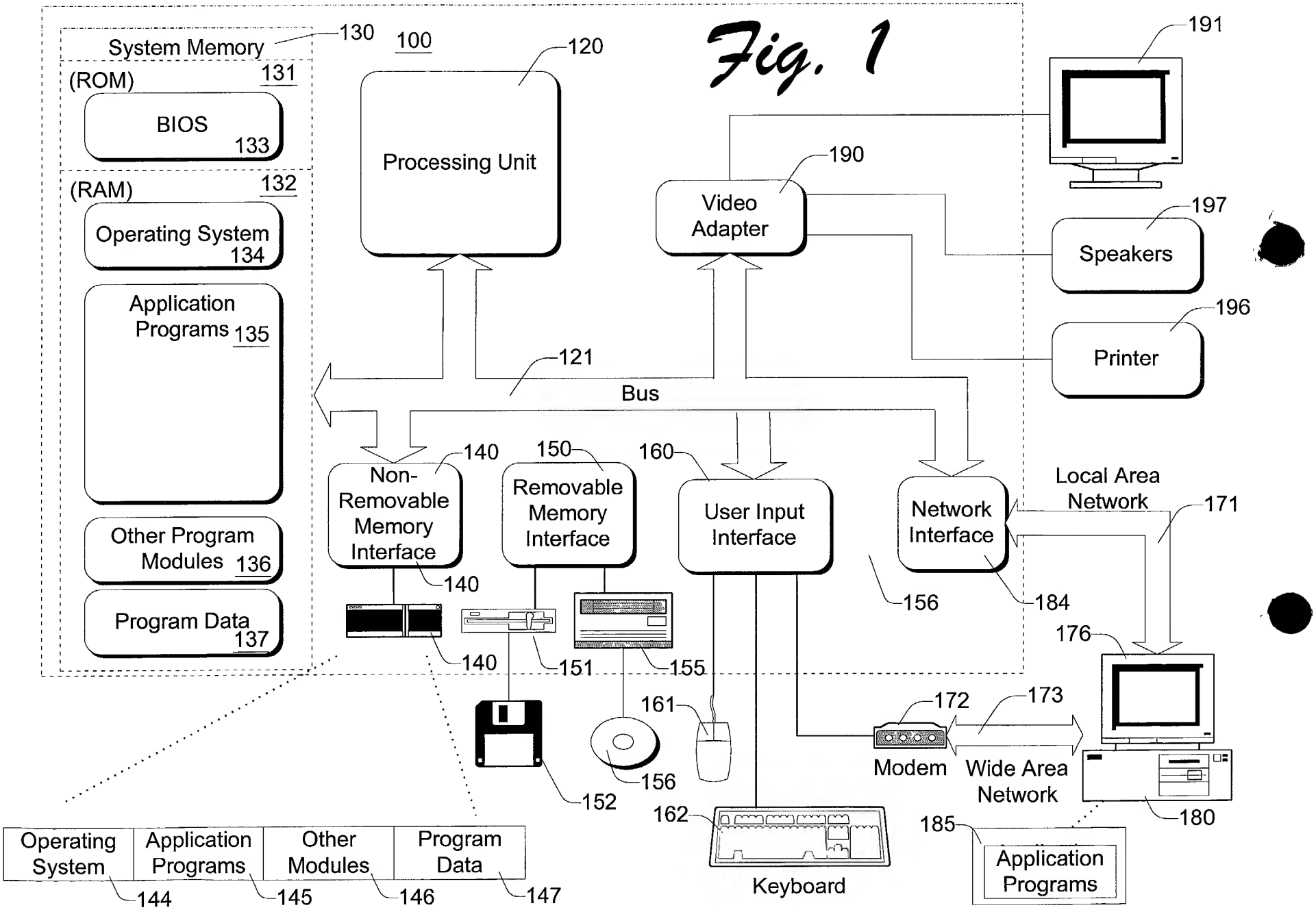
Fig. 1
System Memory
130
100
(ROM)
131
BIOS
133
(RAM)
132
Operating System
134
Application Programs
135
Other Program Modules
136
Program Data
137
Processing Unit
120
Video Adapter
190
191
Speakers
197
Printer
196
121
Bus
Non-Removable Memory Interface
140
Removable Memory Interface
150
User Input Interface
160
Network Interface
Local Area Network
171
156
184
140
140
151
155
161
172
173
176
Modem
Wide Area Network
180
152
156
162
Keyboard
185
Application Programs
Operating System
144
Application Programs
145
Other Modules
146
Program Data
147

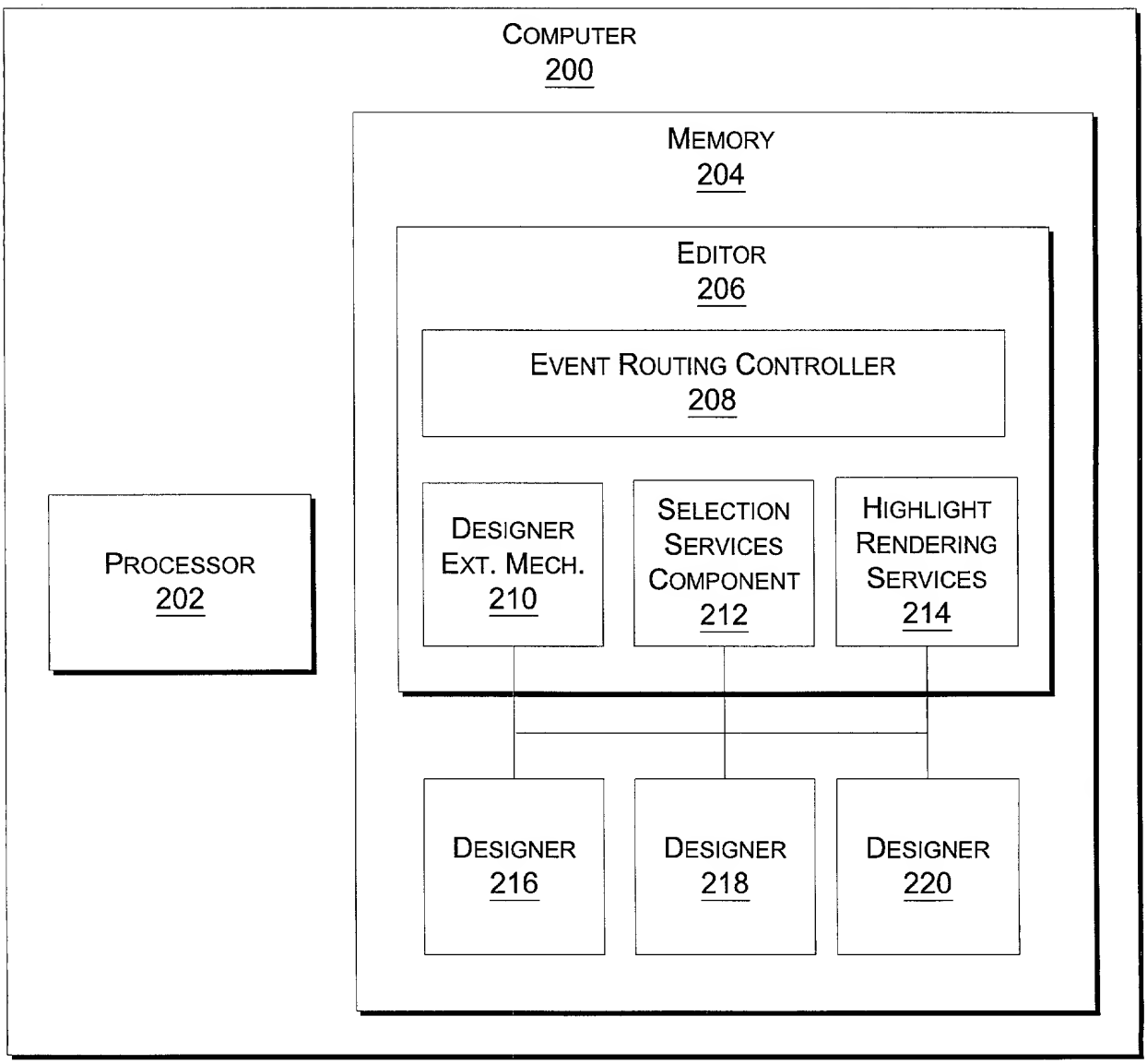
COMPUTER
200
MEMORY
204
EDITOR
206
EVENT ROUTING CONTROLLER
208
PROCESSOR
202
DESIGNER EXT. MECH.
210
SELECTION SERVICES COMPONENT
212
HIGHLIGHT RENDERING SERVICES
214
DESIGNER
216
DESIGNER
218
DESIGNER
220

Fig. 2

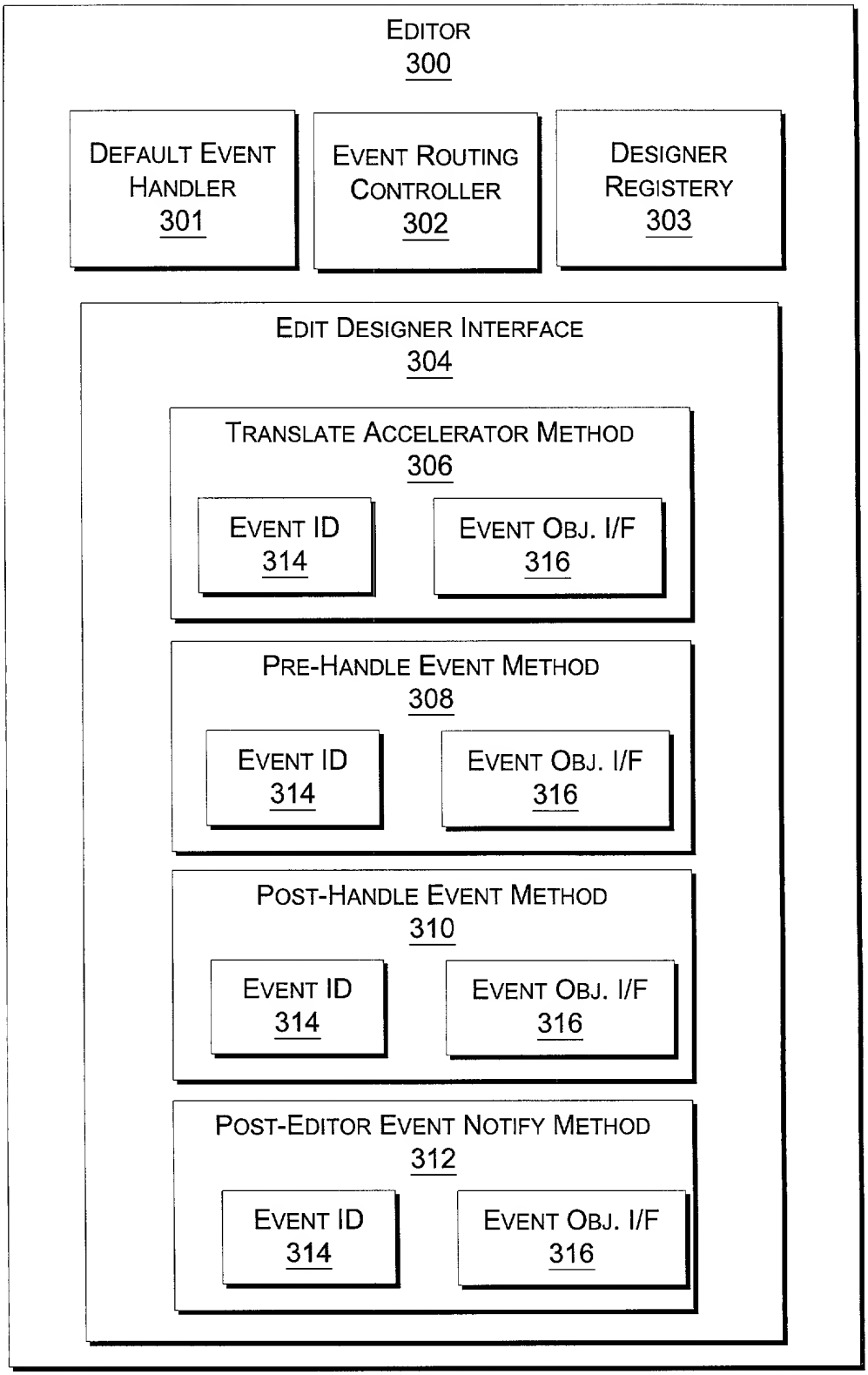
EDITOR
300
DEFAULT EVENT HANDLER
301
EVENT ROUTING CONTROLLER
302
DESIGNER REGISTERY
303
EDIT DESIGNER INTERFACE
304
TRANSLATE ACCELERATOR METHOD
306
EVENT ID
314
EVENT OBJ. I/F
316
PRE-HANDLE EVENT METHOD
308
EVENT ID
314
EVENT OBJ. I/F
316
POST-HANDLE EVENT METHOD
310
EVENT ID
314
EVENT OBJ. I/F
316
POST-EDITOR EVENT NOTIFY METHOD
312
EVENT ID
314
EVENT OBJ. I/F
316

Fig. 3

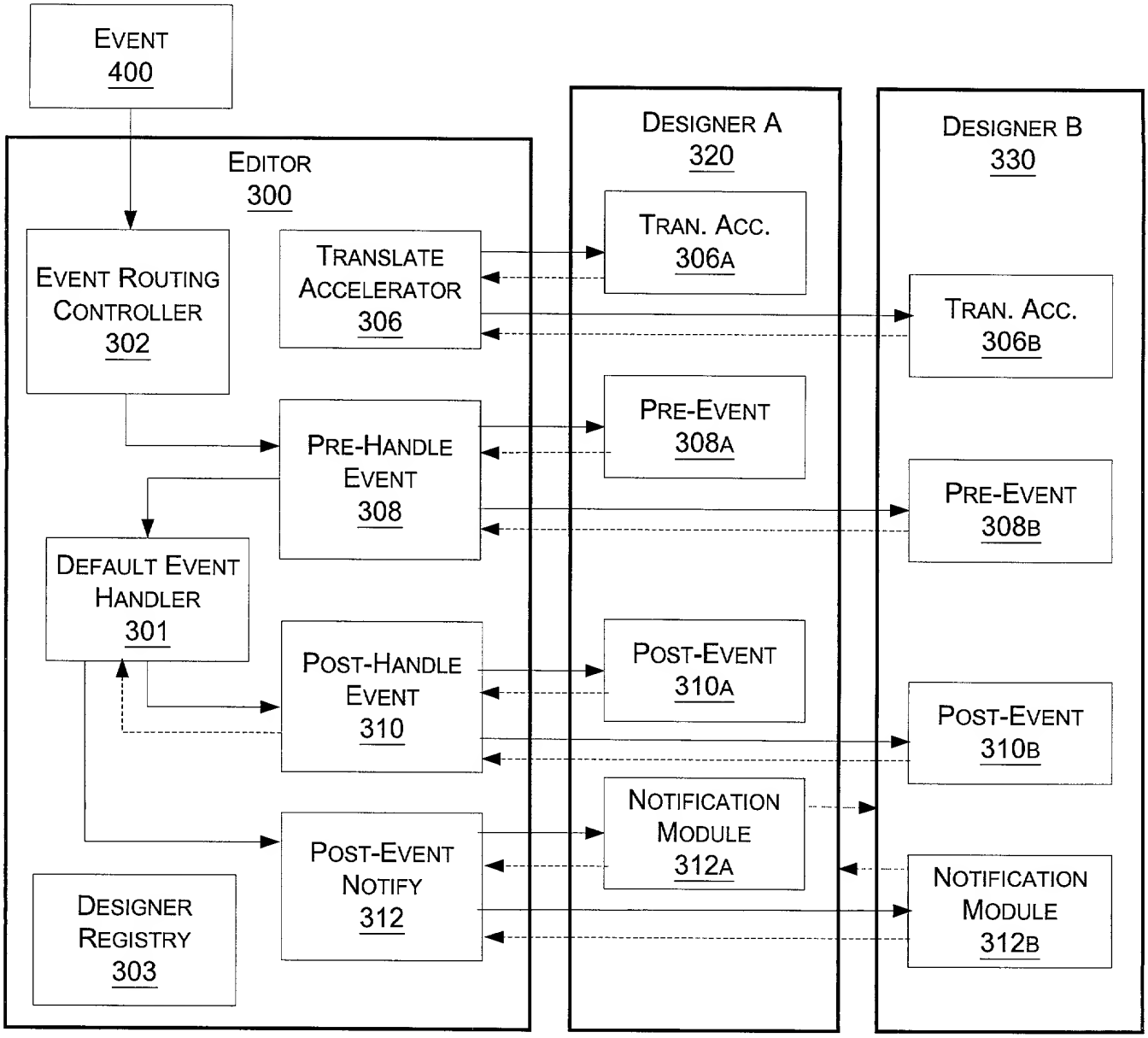
EVENT
400
EDITOR
300
EVENT ROUTING CONTROLLER
302
TRANSLATE ACCELERATOR
306
PRE-HANDLE EVENT
308
DEFAULT EVENT HANDLER
301
POST-HANDLE EVENT
310
POST-EVENT NOTIFY
312
DESIGNER REGISTRY
303
DESIGNER A
320
TRAN. ACC.
306A
PRE-EVENT
308A
POST-EVENT
310A
NOTIFICATION MODULE
312A
DESIGNER B
330
TRAN. ACC.
306B
PRE-EVENT
308B
POST-EVENT
310B
NOTIFICATION MODULE
312B

Fig. 4

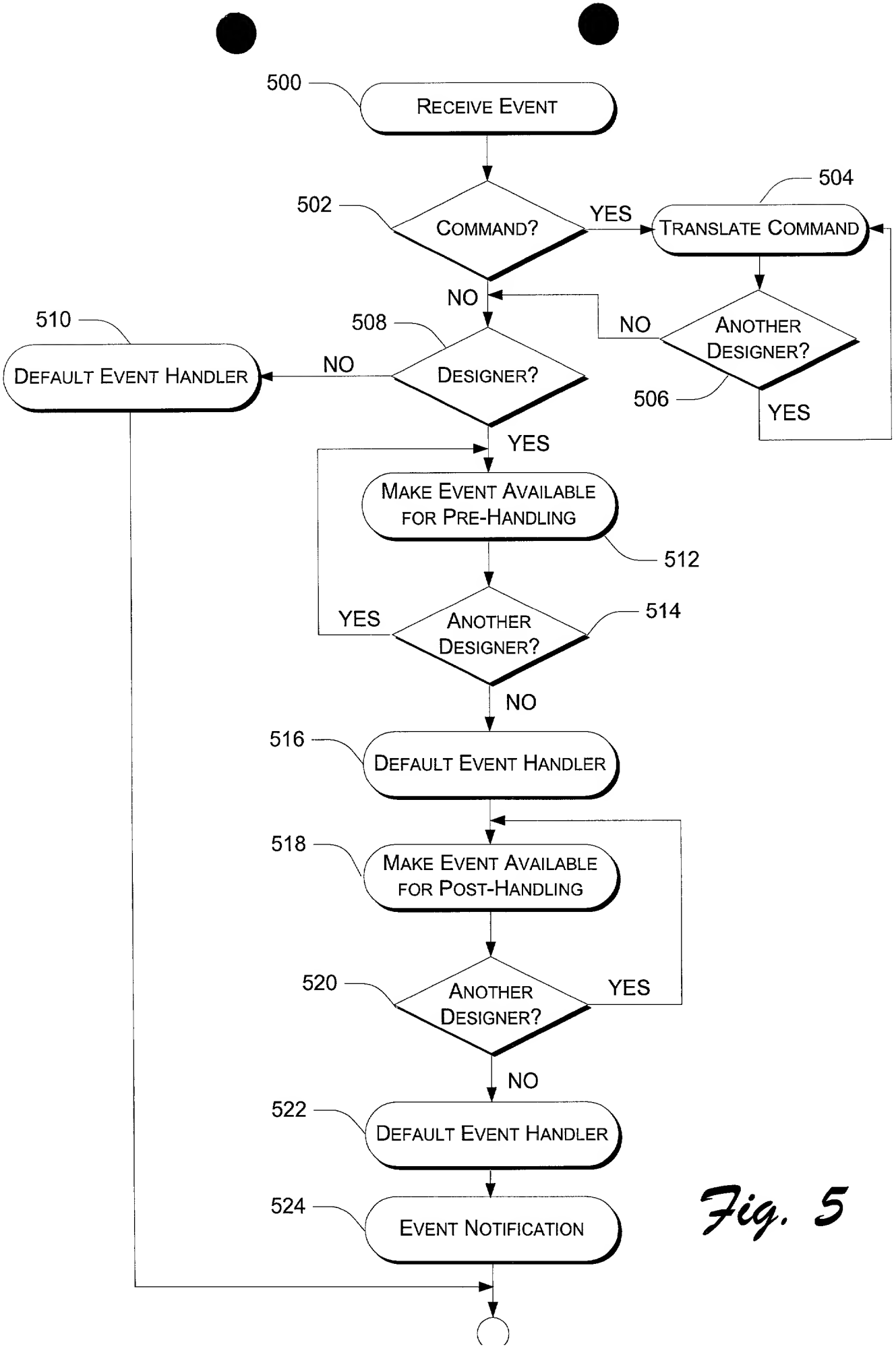
500
RECEIVE EVENT
502
COMMAND?
YES
504
TRANSLATE COMMAND
NO
NO
ANOTHER DESIGNER?
506
YES
510
DEFAULT EVENT HANDLER
508
NO
DESIGNER?
YES
MAKE EVENT AVAILABLE FOR PRE-HANDLING
512
YES
ANOTHER DESIGNER?
514
NO
516
DEFAULT EVENT HANDLER
518
MAKE EVENT AVAILABLE FOR POST-HANDLING
520
ANOTHER DESIGNER?
YES
NO
522
DEFAULT EVENT HANDLER
524
EVENT NOTIFICATION

Fig. 5

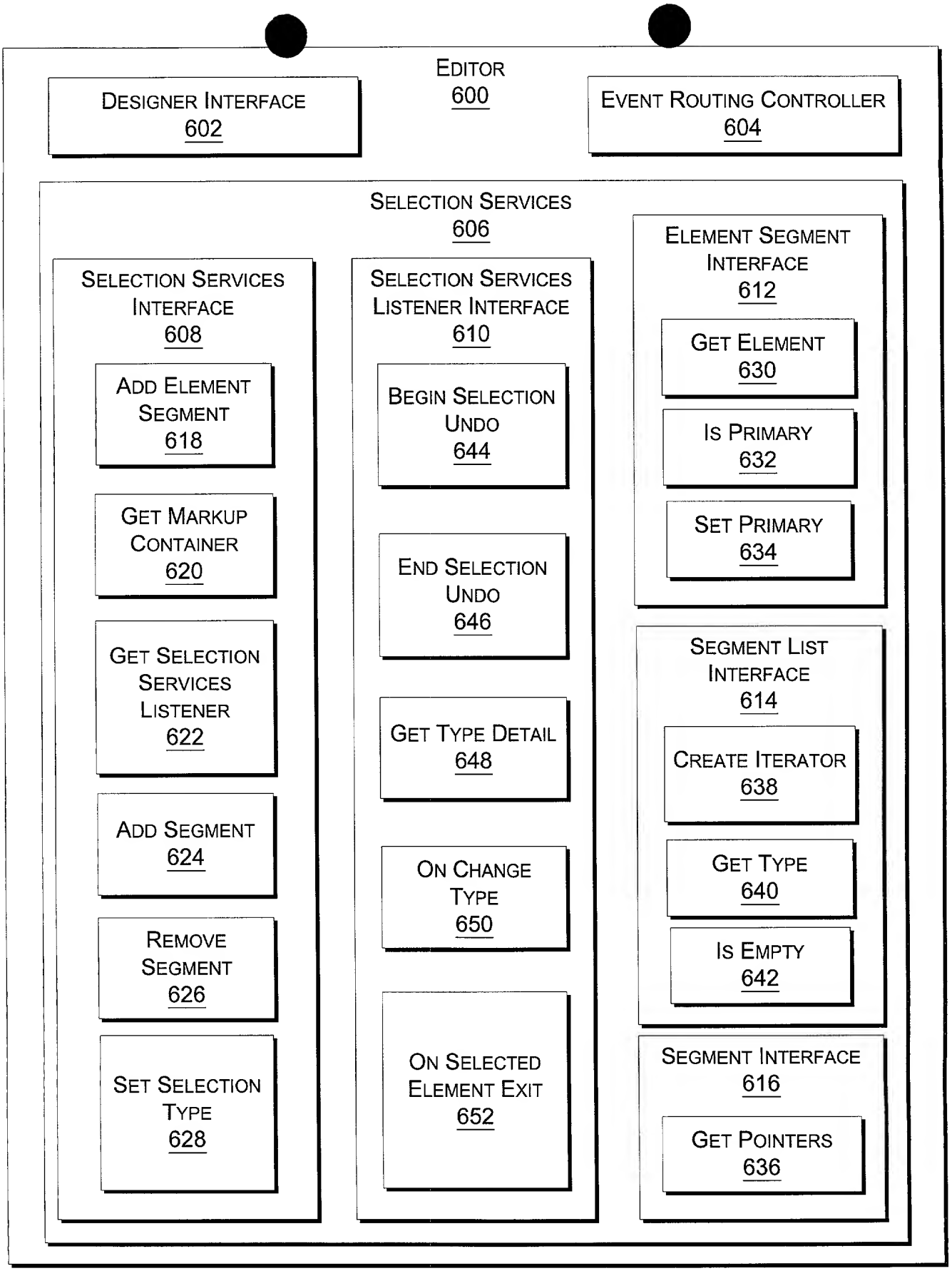
EDITOR
600
DESIGNER INTERFACE
602
EVENT ROUTING CONTROLLER
604
SELECTION SERVICES
606
SELECTION SERVICES INTERFACE
608
ADD ELEMENT SEGMENT
618
GET MARKUP CONTAINER
620
GET SELECTION SERVICES LISTENER
622
ADD SEGMENT
624
REMOVE SEGMENT
626
SET SELECTION TYPE
628
SELECTION SERVICES LISTENER INTERFACE
610
BEGIN SELECTION UNDO
644
END SELECTION UNDO
646
GET TYPE DETAIL
648
ON CHANGE TYPE
650
ON SELECTED ELEMENT EXIT
652
ELEMENT SEGMENT INTERFACE
612
GET ELEMENT
630
IS PRIMARY
632
SET PRIMARY
634
SEGMENT LIST INTERFACE
614
CREATE ITERATOR
638
GET TYPE
640
IS EMPTY
642
SEGMENT INTERFACE
616
GET POINTERS
636

Fig. 6

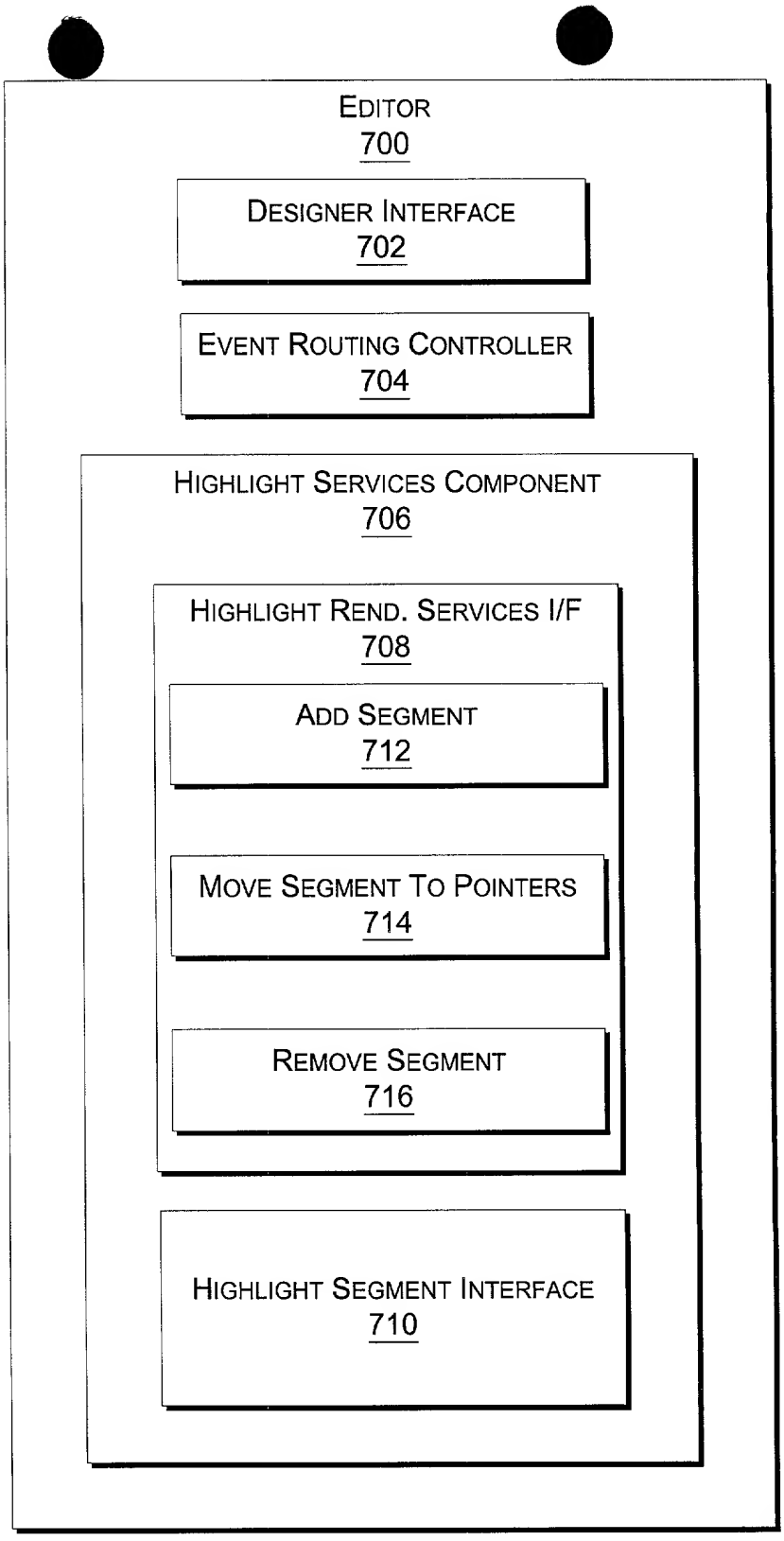
EDITOR
700
DESIGNER INTERFACE
702
EVENT ROUTING CONTROLLER
704
HIGHLIGHT SERVICES COMPONENT
706
HIGHLIGHT REND. SERVICES I/F
708
ADD SEGMENT
712
MOVE SEGMENT TO POINTERS
714
REMOVE SEGMENT
716
HIGHLIGHT SEGMENT INTERFACE
710

Fig. 7